The Inter4QL Interpreter*

Patryk Spanily

11 October 2012

*Supported by the Polish National Science Centre grant 2011/01/B/ST6/02769.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 4QL Language . . . 3

2 4QL Interpreter Design **3**
2.1 Grammar of 4QL . . . 4
2.2 Requirements on the Interpreter . . . 6
2.3 Tools Used . . . 7
2.4 The Class Diagram . . . 8

3 The Implementation of the Interpreter **8**
3.1 Lexical and Syntactic Analysis of the Input Data . . . 8
3.2 Types of Modules . . . 10
3.3 Modules Defined by Sets of Relations, Rules and Facts . . . 10
3.4 External modules . . . 11
3.5 Internal Modules . . . 11
3.6 Operations on Relational Database . . . 11

4 User Documentation of the Interpreter **13**
4.1 Compiling from Sources . . . 14
4.2 Using the Interpreter . . . 14
4.3 Folders and File Structure in the Project . . . 17

5 Further Development of the Interpreter **18**
5.1 Expanding the Terminal Capabilities of the Interpreter . . . 18
5.2 Extending Internal Modules . . . 20
5.3 External modules . . . 22
5.4 Optimizations . . . 24

A 4QL Language Grammar **25**

B Interpreter Command Line **27**

C 4QL programs examples **28**

D The Structure of XML Files Interpreted by Inter4QL **29**

1 Introduction

This paper presents Inter4QL – the interpreter of the rule-based query language 4QL with negation in bodies and heads of rules, belonging to the family of DATALOG$^{\neg\neg}$ languages.

The paper is organized as follows. The first section motivates this paper and discusses main properties of the 4QL language. Section 2 provides the main objectives and capabilities that the interpreter should achieve. It also presents technologies used to develop the application and describes the class diagram. Section 3 discusses the implementation of various parts of the system. Section 4 is devoted to user documentation of the interpreter. The last section gives a brief survey of modifications that can be included in future versions of the application. Appendix A provides Extended Backus-Naur Form grammar of the 4QL language. Appendix B lists the interpreter commands with explanations how to use them. In Appendix C examples of 4QL programs can be found. The last appendix presents the structure of XML files that can be interpreted by Inter4QL.

1.1 Motivation

This work is a companion paper to the application included in the master thesis [4]. The main result of the thesis was the Inter4QL interpreter and this paper is focused on the way the application was developed and how to properly use it.

1.2 4QL Language

4QL is a language that belongs to the family of DATALOG$^{\neg\neg}$ language, introduced in [2]. It uses a four-valued semantics, in which in addition to the "truth" and "falsity" there are also additional values for "unknown" and "inconsistency". An important feature of 4QL is its ability to directly address problems related to inconsistencies and lack of information. These issues are crucial in a number of areas where the closeness of the world is typically assumed (called the Closed World Assumption and denoted by CWA). According to CWA all the facts for which there is no information about, are by default assigned the value "false".

In many applications, including Semantic Web technologies or multi-agent systems, CWA does not necessarily work. Those fields usually accept the Open World Assumption, where the facts about which there is no information, are assigned the value "unknown".

More information about the language 4QL and the group working on it can be found at [11].

2 4QL Interpreter Design

This section introduces a design of a 4QL language interpreter. We will present the main objectives of the system as well as technologies that have been used for implementation. In addition, the class diagram will be fully described.

2.1 Grammar of 4QL

This section presents the schema of programs that are written using 4QL syntax grammar.

Built-in Types

The following data types can be used in the interpreter:

- `literal` - alphanumerical starting with a lowercase letter
- `integer` - decimal number without fractional part (with '–' in front or without)
- `real` - decimal number with fractional part (with '.' sign as decimal sign)
- `logic` - logical value from the set {true, false, unknown, incons}
- `date` - date written in format YYYY-MM-DD where YYYY is year, MM is month (with leading zero) and DD is day (with leading zero)
- `datetime` - date with time in format YYYY-MM-DD HH-II where HH stands for hour (24-hour clock, with leading zero) and II stands for minutes (with leading zero)
- `variable` - additional type (written as alphanumerical starting with a uppercase letter) which can be used only in rules and queries.

File format

A simple schema of the program file written in 4QL is presented in Listing 1.

```
external:
    \\ list of external modules
module X:
    domains:
        \\ declaration of domains for module X
    relations:
        \\ declaration of relations for module X
    rules:
        \\ declaration of rules for module X
    facts:
        \\ declaration of facts for module X
end.
module Y:
    \\ declaration of next module Y
end.
```

Listing 1: simple schema of a 4QL source code file.

Double backslash \\ in 4QL grammar is used for indicating a comment (everything after \\ until the end of the line is ignored by the interpreter). In each program (each file) there can be an unlimited number of modules (but each name of a module must be unique). External modules are described briefly in Subsection 3.4.

Full grammar of 4QL in Extended Backus-Naur Form is provided in Appendix A.

Declaration of Domains

In the section of domain declarations an *alias* to a name of a certain type can be given. Those *aliases* are useful in relationship declarations (relationship arguments can be of the same type, so using aliases helps to distinguish different parameters). Each domain declaration is a set of two alphanumerical strings starting from a lowercase letter (where the first one is a name of a built-in type, and second is the name of an alias) ended with a '.' (dot sign). An example is given in Listing 2.

```
domains:
  literal name.
  integer height.
  date dateOfBirth.
```

Listing 2: exemplary domains declaration.

Declaration of Relations

Relation declarations consist of relation names together with a list of types (or type aliases as defined in the domain declaration section). Each relationship should have a unique name and be ended with a '.' (dot sign). Example of relation declarations can be found in Listing 3.

```
relations:
  tall(name).
  hasHeight(name, height).
  hasBirthdate(name, dateOfBirth).
```

Listing 3: exemplary declaration of relations.

Declaration of Rules

In the 'rules' section definitions of all the rules for a given module must be provided. All of them (headers and bodies) should be compatible with the set of relations defined in the previous section. In rules values of type `variable` can be used. Each declaration of a rule must end with a '.' (dot sign). Listing 4 contains a sample section of rules declaration (with only one rule declared).

```
rules:
    tall(X) :- hasHeight(X, Y), math.gt(Y, 180).
```

Listing 4: exemplary declaration of a rule.

Relation `math.gt` in the example is a relation from a built-in module math with the intuitive meaning "*greater than*".

Declaration of Facts

This part is responsible for defining the initial database of facts (the initial set from which the reasoning starts). Each fact is defined by the name of a relation (from the set of relations defined in that particular module) and the list of constants of respective types (that must be compatible with the set of relation of that module). Listing 5 contains an example of the base facts.

```
facts:
  hasHeight(patryk, 195).
  hasBirthdate(patryk, 1988-05-20).
```

Listing 5: exemplary declaration of facts.

Declaration of External Modules

In this part all the external modules (more about external module can be found in Subsection 3.4) need to be defined. Each line is a declaration and starts with the name of an external module, type of that module and a list of parameters. Each declaration ends with a . (dot sign). An example of the external modules declaration is shown in Listing 6.

```
external:
  externalData xml("file_with_facts.xml").
```

Listing 6: declaration of external modules example

2.2 Requirements on the Interpreter

Interpreter analyzes the source code and after processing the data it executes analyzed fragments. In the case of the application described in this paper, by executing the fragments of source code we understand reasoning based on logical rules in accordance to the well-supported model calculation algorithm (described in [2, 3]). However, apart from this functionality, the interpreter should give additional features and that is the main subject of this subsection.

Interpretation of 4QL Programs

The basic functionality of the interpreter is loading source codes written in the 4QL language. In the case of an incorrect attempt to load a program (various syntactic errors, lexical errors, incorrect references inside rules, etc.) the system should write a human understandable list of error messages (allowing the author to fix mistakes). After loading all modules of a correct program, all of these modules should remain in the environment and the interpreter should allow to load another program (an error should be presented if one would like to load a module with the same name as a module already existing in the environment).

The Calculation of Well-Supported Model

After loading the program, the interpreter should check that references to the external literals in all modules do not contain recursion. If they do, the interpreter should display the information about the list of the modules that have recursive references to external literals. The graph of module references should be calculated by the interpreter (so the reasoning should start from the lowest layer). The last phase after loading a program and before user interaction depends on the use of the algorithm calculating the well-supported model in accordance with the previously computed order of reasoning.

Responding to User Queries

After calculating the well-supported model for each module from a loaded program, the interpreter should allow to query about the facts that are in the knowledge base. In addition to its simplest form, in which query relates to a fact (as described by the name of module, the name of fact and a list of values), the interpreter must support queries with variables – the answer should be the list of all the facts of a particular module which fit the query.

Saving the Results in the Form of an XML File

After loading the program (or programs), the interpreter should give the opportunity to write all or just part of the knowledge base to a file in an XML format.

Loading Data from Modules in the form of XML Files or External Knowledge Bases

The interpreter must be able to read the data stored in the XML file or external database files. Defining new types of files or databases should not be complicated in further development of the interpreter.

2.3 Tools Used

The interpreter has been written applying the following tools:

- *C++* language with the use of *STL*
- *Flex - The Fast Lexical Analyzer* library (more can be found at [6])
- *GNU Bison* library (more can be found at [5])
- *TinyXML* library (more can be found at [7])
- *Doxygen* - documentation generator (more can be found at [8]).

A more detailed description of the use of these tools is provided in Section 3.

2.4 The Class Diagram

Figure 2.4 shows the class diagram for Inter4QL. The diagram is divided into four main parts (color coded) and additional two helper classes (marked as white classes). These are the main functionalities of different parts of the application:

- *yellow* colored - *Application* class being the main interpreter class
- *green* colored - classes that do lexical and syntactic analysis and correctness checking
- *violet* colored - classes that deal with modules of different types
- *red* colored - classes dedicated to the establishing connection to the database.

Additional classes with white color deal with:

- *VariableSpace* class - storing all values occurring in the programs loaded into environment
- *Output* class - printing the information on the screen.

3 The Implementation of the Interpreter

This section presents the implementation of the individual parts of the system. Subsection 3.1 describes the scanning and parsing part of the Inter4QL and the command-line interpreter. Subsection 3.2 explains how different types of modules are stored in application. The last subsection covers methods of reasoning using information stored in the knowledge base.

3.1 Lexical and Syntactic Analysis of the Input Data

Frequently, the first screen of interpreter is a blank line and a command prompt. Communication with the system is made by typing commands into the terminal and the results can be seen (usually immediately) on the screen. The first part of the interpreter application should recognize commands entered by the user. From the beginning, the line entered in the standard input should be divided

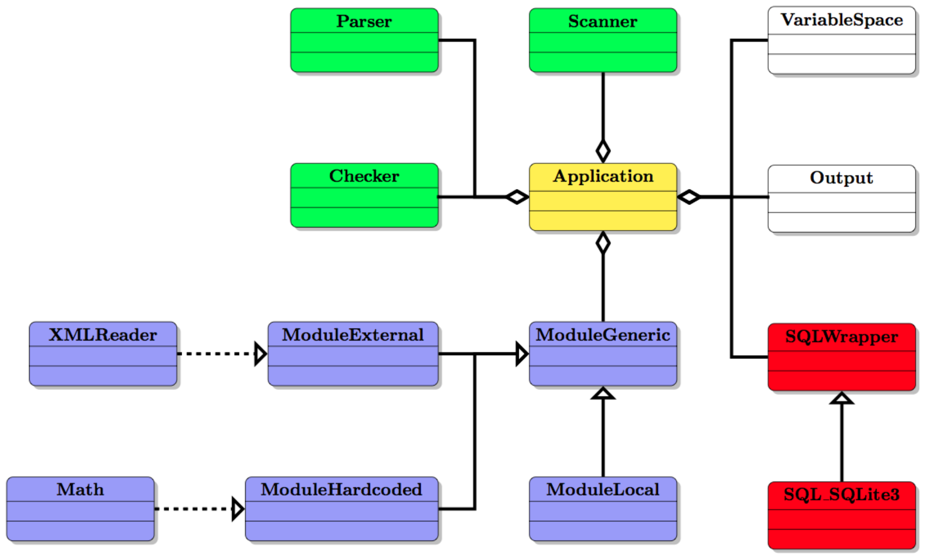

Figure 1: Class diagram.

into appropriate tokens. Inter4QL uses *Flex - The Fast Lexical Analyzer* as a library that is responsible for that part (the full documentation of flex library can be found at [6]). *Flex* library allows the user to create a scanner with only a list of regular expressions (*Flex* uses regular expression to split the string into tokens called lexems). The tokens can also have additional values. For example, `INTEGER` token can also store its integer value. A properly prepared file with regular expressions[1] is then translated by flex into the C++ code.[2] The same regular expressions are used in lexical analysis of the command-line interpreter, as well as to analyze the source code of programs written in 4QL.

After the stage of translating the text into tokens, the interpreter needs to check whether the lexem string is defined correctly. Inter4QL uses *GNU Bison* library [5] for this part of input processing. *GNU Bison* library simplifies writing parsers - one only needs to define the grammar of the language using productions, where terminal expressions are lexems or lexem strings and the library will generate appropriate files in the C language. The full code of 4QL grammar written in the *GNU Bison* language is provided in the file `LineParser.y` in the directory with application sources.

Files automatically created by the library are:

`LineParser.tab.c` and `LineParser.tab.h`.

To maintain consistency in the project there is an additional `Parser` class (with

[1]In Inter4QL flex definition file name is Scanner.l.

[2]In Inter4QL `FlexLexer` class methods are defined in file `lex.yy.cc`, and `FlexLexer` class is defined in `FlexLexer.h`.

definition in `Parser.h` and implementation in `Parser.cc`). It uses the *GNU Bison*'s generated source codes in C.

A part of the application described in this section is indicated on the diagram in Subsection 2.4 with the green color. The way how to extend the interpreter with additional commands will be described in Subsection 5.1.

3.2 Types of Modules

In the previous section three types of modules have been introduced:

- modules defined by the program code in language 4QL
- external modules - external files with knowledge base facts
- internal modules - modules built into the interpreter environment (e.g., `math`).

All three types of modules share a common interface (the definition can be found in class `ModuleGeneric` inside `ModuleGeneric.h` file), but they differ significantly in implementation. In this subsection the differences will be described. For the way of expanding the capabilities of the interpreter by new internal and external modules see Subsections 5.2 and 5.3, respectively. This part of the system has been marked on the classes diagram on Figure 2.4 in Subsection 2.4 with *violet* color.

Generic Interface

From the perspective of the `Application` class which deals with the entire interpreter application, all types of modules are the same. Through the use of a single interface, the flow of information between classes is the same regardless of implementation of that particular module. This has been achieved by using type casting `(ModuleGeneric *)`. All types of modules have a set of relations and a set of facts. Facts can be described as an initial set of facts (in the case of the defined modules) or the final set of facts (for external modules). Both facts and relations are stored in a database by using a generic interface `SQLWrapper`.

The following sections will cover the basic differences between different types of modules.

3.3 Modules Defined by Sets of Relations, Rules and Facts

Code defined modules are basic type of module in the Inter4QL interpreter (other types of modules are used as a “helpers” only). They are created as instances of `ModuleLocal` class (and then cast to a pointer to a generic interface `ModuleGeneric`). They are the only modules that have rules and domains, thus the only one where the reasoning is made. After the file parsing stage of a 4QL program, all rules are kept in appropriate structures (`std::vector<Rule *>`). `ModuleLocal` is the only type of module where interpreter can make a reasoning with the well-supported model calculation algorithm. All phases of reasoning are held on the database side (more about the way it is implemented can be found in Subsection 3.6).

3.4 External modules

External modules are responsible for connections to external data sources. They have only finite relations and facts that will never change since the moment they are read from the external source file (because no reasoning is made on external modules). The way they are created as instances of ModuleExternal is described in Subsection 5.3.

3.5 Internal Modules

Internal modules are responsible for the constructs that are difficult to express by 4QL rules. The simplest example is the internal math module, which provides a few relations like "greater than", "lower than", etc. Another example is the module which allows users to convert data types (e.g., `INTEGER` to `REAL`). Such structures can be expressed as `DATALOG` rules, but only in a very inefficient way. However, there is a drawback of internal modules – the interpreter calculates all the possible facts before the start of reasoning. Optimization, which can prevent the above disadvantage is described in Subsection 5.4. The way of creating new `ModuleInternal` instances is described in Subsection 5.2.

3.6 Operations on Relational Database

Database management system is a very important part of Inter4QL system since the reasoning is based on SQL statements. Inter4QL does not impose the use of any specific database management system but its current version uses *SQLite3* [9]). A generic class `SQLWrapper` (from `SQLWrapper.h` file) allows implementation of connection to any database management system (*PostgreSQL*, *Oracle*, *MS SQL*, etc.). The current version implements the connection inside the files `SQL_SQLite.cc` and `SQL_SQLite.h`. *SQLite3* gave the possibility to work in-memory, so no additional configuration or external database management system is needed for Inter4QL to work. This section will describe the way of keeping facts and the way the reasoning is made (the well-supported model calculation algorithm) in a *SQLite3* database. The part of the application described in this section is indicated on the diagram in the Subsection 2.4 with red color.

Relations and Facts in Database

SQLite3 is a simple database management system in which there is no distinction among data types. Everything is stored as a string (a sequence of characters). This causes a slow down of the system, but it also simplifies the way the tables and queries are created in Inter4QL.

Each relation named {`relation_name`} defined in a module called {`module_name`} and n parameters will correspond to a database table with the name {`module_name`}_{`relation_name`} with $n+2$ columns: `param1`, `param2`, ..., `param`n, `is_true`, `is_false`. The first n parameters correspond to the arguments of relation, while the last two correspond to the value in the tetravalent logic (it holds information about the fact that the parameter stored in the first n columns). Here is an example of relationship `hasHeight` in the module `info`:

`info_hasHeight`			
`param1`	`param2`	`is_true`	`is_false`
`patryk`	`185`	`1`	`0`
`marcelina`	`165`	`1`	`0`
`patryk`	`165`	`0`	`1`

The table shows that in the database there is an information on `patryk` height (we know that the fact `hasHeight(patryk, 185)` is true and that fact `hasHeight(patryk, 165)` is false) and `marcelina` height (there is an information only about a true fact: `hasHeight(marcelina, 165)`).

In addition to the previously described tables in the database for each table named {`module_name`}_{`relation_name`} with n+2 columns there exists adequate views with n columns with names:

{`module_name`}_{`relation_name`}`_true`

and

{`module_name`}_{`relation_name`}`_false`.

Those views are created by selecting only the positive facts, and only negative facts (it can happen that the fact is inconsistent, so it will exist in both of these views).

In addition, there are also views

{`module_name`}_{`relation_name`}`_only_true`

and

{`module_name`}_{`relation_name`}`_only_false`,

which are similar to the previous ones, but do not contain inconsistent facts. Those views are useful in further phases of reasoning.

Reasoning about Database Facts

In the previous section the way that facts are kept in the database was shown. This section will present the algorithm which was used for reasoning on the data stored in a relational database.

In the first two phases of the algorithm presented in [2] least Herbrand models are computed. In order to do this it needs to be checked whether there is a rule, whose body is factually true from the database and the fact being in this rule header does not yet exist in the database. This action needs to be repeated until the point at which none of the rules will generate a new fact.

How can it be checked if the rule generates new facts? A proper SQL statement should be prepared. The easiest way to show a method of preparing such queries is example based on module info, as shown in Listing 7.

```
tallBoy(A) :- boy(A), hasHeight(A,B), math.hd(B,180).
```

Listing 7: exemplary rule.

Since the assumption is made that this rule correctly refers to relations, it follows that in the database there are at least tables: `info_tallboy`, `info_boy`, `info_hasheight`, `math_ht` (and, of course, adequate views to these tables). To check whether this rule generates new fact (or facts) we need to check links between tables based on a rule expressed in the natural language:

"*if a boy* `A` *has height* `B`*, and* `B` *is greater than* `180` *then* `A` *is a tall boy*".
The SQL query for that rule is shown in Listing 8.

```
select t1.param1 from
  info_boy_true t1, info_hasheight_true t2, math_ht_true t3 where
  t1.param1 = t2.param1 and t2.param2 = t3.param1 and t3.param2 =
      180;
```

Listing 8: SQL query for rule from Listing 7.

If this query returns any new rows, it means that the column in each returned row (denoted as param) `tallBoy(param)` is true. Now it is sufficient to check whether in that returned set of rows there exists a fact that was not present in the database. In the case that none of the rules generates new facts, the least Herbrand model was calculated.

The second phase of reasoning is almost the same - the set of rules that take part in the algorithm must be reduced by removing rules whose headers in the first phase have been marked as inconsistent.

In the third phase of the algorithm two queries should be prepared: the first one, which asks for true facts (inconsistent included) and the second, which reasons only throughout true facts (inconsistent excluded). Those two queries result sets need to be subtracted with the SQL command `except`, and the same as in the previous phases this action continues until the moment that no new facts are generated.

4 User Documentation of the Interpreter

This section deals with user documentation of the interpreter:

- installing (compiling) – see Subsection 4.1
- the way of using the Inter4QL application – see Subsection 4.2

- the structure of files and folders in the project – see Subsection 4.3.

4.1 Compiling from Sources

Inter4QL project has `Makefile` file, which is responsible for automatic compilation of the application. In the Linux family systems or Mac OS X, in order to compile from sources a *g++* compiler is needed (the author used *g++* version 4.5.2). In the systems of the Windows family the user needs to have a *MinGW* package (*Minimalist GNU for Windows* [10]) and compile the project within that environment. In addition, to compile the project the following libraries are needed:

- *Flex* - version 2.5.35 or newer
- *GNU Bison* - version 2.4.2 or newer.

The binary version for Windows family operating systems is included with the project sources and is available in the folder *bin/*. For the Linux and Mac OS X version user must manually build the application.

4.2 Using the Interpreter

This section will describe the standard mode of using the interpreter environment of Inter4QL. The interpreter is started by running the command:
`inter4ql.exe` (for the Windows family operating system) or
`./inter4ql` (both for Linux and Mac OS X family operating systems)
in the directory of the executable application file. On startup the screen should display information about the version of the application and a command line prompt. The list of all interpreter commands can be found in Appendix B. In the following sections we will describe the most important commands of the interpreter.

Importing 4QL Programs

The basic function of Inter4QL is to interpret programs from the source code files. Command `import` is responsible for reading external 4QL source code files. After import, the user needs to prepare the path of the file he or she wants to import in quotes and end the command with '`.`' (dot sign). An example of the import command can be found in listing 9.

```
#import "filename.4ql"
Program loaded!
```

Listing 9: example of file importing.

If the source program contains errors, instead of the string `Program loaded!` the interpreter will print the corresponding error on the screen. After executing this command, all local modules will have their unique well-supported models calculated.

Querying the Information Contained in the Knowledge Base

After loading modules (i.e., after the hidden reasoning stage), the user can start querying the interpreter about the data in the knowledge base. In order to ask for the value of the fact the name of the module, the name of the relation and all their parameters must be entered and the whole command must be ended by dot sign (see Listing 10).

```
# moduleName.relationName(param1, param2, ..., paramN).
```

Listing 10: example of knowledge base query.

In Listing 10 param$n$ is n-th parameter of the fact with appropriate type or a variable (alphanumeric word that starts with a capital letter).

Listing 11 contains examples of the implementation of queries to the knowledge base (on a slightly expanded set of data from the second example in Appendix C).

```
# data.tallBoy(A).
results:
        data.tallBoy(tomek) : inconsistent
# data.hasHeight(A,B).
results:
        data.hasHeight(marcelina, 165) : true
        data.hasHeight(tomek, 190) : true
        data.hasHeight(tomek, 180) : false
# data.hasHeight(tomek, A).
results:
        data.hasHeight(tomek, 190) : true
        data.hasHeight(tomek, 180) : false
# data.hasHeight(patryk, A).
results:
no results
```

Listing 11: more examples of knowledge base queries.

Saving the Database in the SQLite3 File Format

Inter4QL can save the entire database as a file in *SQLite3* database format. This gives the possibility to produce queries in SQL and provides additional ways of presenting the computed data. In order to save the knowledge base command `save` must be used with the filename in quotes ended by a dot sign. An example is shown in Listing 12.

```
# save "sqlite3.db".
saving database to: "sqlite3.db"
```

Listing 12: saving environment to a database file.

In case of any problems, Inter4QL prints on the screen the relevant information about the cause of the error.

Saving Modules as XML Files

In addition to saving the entire knowledge base as a database file, the interpreter can also save a unique well-supported model of the module as an XML file. In order to achieve this, the command `save` must be used but with a slightly different syntax. Between the `save` command and filename the user should write the name of module which will be saved as an XML file. As each command of the interpreter it must be ended by '`.`' (dot sign). For an example see Listing 13.

```
# save data "data.xml".
saving module data (as xml) to: "data.xml"
```

Listing 13: saving module to an XML file.

In case of any problems, Inter4QL prints on the screen the relevant information about the cause of the error.

Printing Basic Information about the Modules in the Environment

To print basic information about the modules the command `print` is used. After the keyword the user should provide module name and end the command with '`.`' (dot sign). An example how to print basic information about module `m0` from the first example in Appendix C can be found in Listing 14.

```
# print m0.
module m0:
relations:
    a(literal)
beginning facts:
    m0.a(overloaded (literal)) : true
    m0.a(rested (literal)) : true
    m0.a(success (literal)) : true
rules:
    m0.a(wait (literal)) : true :- m0.a(overloaded (literal)) :
        true
    |       m0.a(resttime (literal)) : true
    m0.a(resttime (literal)) : true :- m0.a(wait (literal)) : true
    m0.a(overloaded (literal)) : false :- m0.a(resttime (literal))
        : true
    m0.a(goodmood (literal)) : true :- m0.a(rested (literal)) :
        true
    |       m0.a(success (literal)) : true
    m0.a(rested (literal)) : false :- m0.a(resttime (literal)) :
        false
```

Listing 14: printing basic module information.

Quitting the interpreter

Command quit is used to turn off the interpreter. After executing this command all the data stored in the memory of the interpreter is released (but all that memory can be saved before quitting by using command save described in previous sections). Listing 15 shows an example of quitting Inter4QL.

```
# quit.
Thanks for using!
RUN SUCCESSFUL (total time: 3m 29s)
```

Listing 15: quitting the interpreter.

4.3 Folders and File Structure in the Project

This section presents the structure of the files in the archive distributed on 4QL.org. The root folder contains the following folders:

- src/ - the source code of the interpreter Inter4QL
- src/sqlite3/ - the source code of the SQLite3 library
- src/tinyxml/ - the source code of the TinyXML library
- tests/ - the set of tests of the interpreter

- `documentation/` - the source code documentation (in html format) automatically generated (based on comments in source code) by doxygen application.

5 Further Development of the Interpreter

The interpreter will be developed in the future and this section presents some of the proposed extensions of the application. Logical design (discussed in Subsection 2.4) helps in modifications or expansions of the system. In the next subsections various ways of how the project can be extended by a programmer will be presented. The final section considers ideas on a variety of modifications and improvements of the interpreter, which were not implemented in the version up-to-date by the time writing this paper.

5.1 Expanding the Terminal Capabilities of the Interpreter

In this section we show how to add new commands to the interpreter on the basis of a command `list` which prints all the loaded modules in the environment. To do this, changes in the files `Types.h`, `Scanner.l`, `LineParser.y` and `Application.cc` needs to be made.

Changes in Definitions of Program Tree

The first step that needs to be done in order to add a new command to the interpreter depends on changing the enumerating type, which indicates what kind of a line is loaded through command-line. This is made by adding a value to the enum `type` inside `line_type` structure in file `Types.h`. In the example shown in Listing 16, the added value will be `LIST_MODULES`, which will mean that the command will be asking for the modules currently loaded in the environment.

```
struct line_type {
    enum type {
        PROGRAM, PRINT, IMPORT, RULE, QUIT, LIST_MODULES
    };
};
```

Listing 16: changes in file `Types.h`.

Changes in the Lexical and Syntactic Analyzers

Changes in the lexical analyzer and syntax analyzer must be interrelated. In the lexical analyzer (or more precisely in the file `Scanner.l`), the relevant instruction (in correct *Flex* syntax) must be added. In the case of detecting a keyword `list` in the input, the scanner must return a relevant type (which in the latter part of this section will be defined in the parser). A particular attention must be paid in order to add a new line to the file `Scanner.l` above the line with the regular expression that could validate the word list (if not, the scanner will validate word list as an alphanumerical, not a `LIST_MODULES` token).

```
list { return Inter4ql::LineParser::token::LIST_MODULES; } // new
[a-z][a-zA-Z0-9\-]* {
  yylval->value = new Value(yytext);
  return Inter4ql::LineParser::token::ID;
}
```

Listing 17: changes in file `Scanner.l`.

The enumeration type `LIST_MODULES` in the moment of editing the file `Scanner.l` does not exist yet, so the next operation is to create the appropriate token in the parser file (in *GNU Bison* syntax). In this example, a new object has to be called `LIST_MODULES` similar to the token that was added to the `type` enum. In the parser's source code (file `LineParser.y`) an entry must be added in the terminal symbols section, as shown in Listing 18.

```
%token LIST_MODULES
```

Listing 18: changes in file `LineParser.y`.

An appropriate entry corresponding to the command must be added into the grammar of the language. By default, all the commands in the interpreter end with a dot that corresponds to the terminal symbol `DOT`. In order to give the parser the capability to read a new command line a production must be added to the nonterminal symbol line (see Listing 19).

```
line : (...)
  |   LIST_MODULES DOT { data.type = LIST_MODULES; }
;
```

Listing 19: additional changes in file `LineParser.y`.

Between the braces in the code, a valid value of variable data (which is returned in main interpreter loop) must be set. In that case the only operation that needs to be done is simply setting the type of line (the same one which was added to the enumeration in struct `line_type` in the beginning of this section).

Changes in the Source code of the Main Module

The last step, which needs to be performed is to edit the method `Program::parse_line()` in the `Application.cc` file. In the body there is a switch (conditional statement) instruction which needs to be extended (example can be found in Listing 20).

```
switch(l->type) { (...)
   case LIST_MODULES:
   {
      this->output->print("available modules:\n");
      for(int i = 0; i < this->modules->length(); i++)
         this->output->print(this->modules[i]->get_name() + "\n");
      break;
   }
}
```

Listing 20: changes in file `Application.cc`.

After making the changes accordingly to previous sections and recompiling the project, a new command `list` is ready to use. Listing 21 shows an example of using this command in a previously prepared interpreter environment:

```
# list.
available modules:
math
module_01
module_xml
```

Listing 21: example of using `list` command.

5.2 Extending Internal Modules

This section will describe a method how to create a new internal module. Accurate description of an internal module can be found in Subsection 3.5. All internal modules should inherit from class `ModuleInteral` which, in turn, is the implementation of the common interface of all types of modules in the system. Special attention should be paid to the class constructor and the method `generate_facts()`. The last part of this subsection describes the way how to add internal module into the environment of the interpreter.

In order to maintain the project properly, all the files with declarations and definitions of additional modules are located in the folder `modules/` and that is the proper place for putting new module source code files. The method of creating new internal modules will be shown on the example of mathematical module (`math`). The full source code of mathematical module can be found in source code files `modules/Math.cc` (implementation) and `modules/Math.h` (definition).

Constructors

Each internal module constructor should collect pointers to class instances of `VariableSpace` and `SQLWrapper`. The first of them gives access to all available values in the program, on which basis it will be possible to generate all the facts. The second pointer allows one to add those facts into the database. The constructor for mathematical module is provided in Listing 22.

```
Math(VariableSpace *_v, SQLWrapper *_sql);
```

Listing 22: constructor definition for module `math`.

In implementation, in addition to call the parent constructor, the following steps must be made:

1. set the appropriate name for the module:

   ```
   this->name = std::string("math");
   ```

2. set the appropriate relation list of that module

   ```
   std::vector<Domain *> *ii = new std::vector<Domain*> ();
      for (int i = 0; i < 2; i++)
         ii->push_back(new Domain(variable_type::INTEGER));
      this->relations = new std::vector<Relation *>();
      // add function gt (greater than), with two integer
          arguments
      this->relations->push_back(new Relation("gt", ii, "math"));
   ```

3. create a module using the method offered by the class responsible for the connection to the database

   ```
   this->sql->create_module(this);
   ```

All the values created in the constructor by the new keyword need to be released by using `delete` keyword in the destructor.

Generating Facts into the Database

Generating facts in internal modules takes place each time a program is loaded (in order to add all the values existing in the program into the variable space), so at the beginning all the facts established in previous call need to be removed. Subsequently, for each relationship defined in the constructor and for all possible values, all facts must be created by using appropriate `add_fact()` method call of the class corresponding to the database link. Sample code that generates all the facts for the function `gt` (greater than) is shown in Listing 23.

```
void Math::generate_facts() {
    for (int i = 0; i < this->relations->size(); i++)
        this->sql->delete_from_table(this->relations->at(i)->
            sql_name());
    std::map<int, Value*> *is = this->variable_space->
        get_integer_space();
    std::map<int, Value*>::iterator it1, it2;
    for (it1 = is->begin(); it1 != is->end(); it1++) {
        for (it2 = is->begin(); it2 != is->end(); it2++) {
            std::vector<Value *> * v = new std::vector<Value *>();
            v->push_back(this->variable_space->get_pointer(it1->
                first));
            v->push_back(this->variable_space->get_pointer(it2->
                first));
            if (it1->first > it2->first) {
                this->sql->add_fact(new Fact("math", "gt", v));
            } else {
                this->sql->add_fact(new Fact("math", "gt", v, 0, 1)
                    );
            }
            delete v;
        }
    }
    return;
}
```

Listing 23: example of fact generation process.

Connecting the Internal Module into the Interpreter Environment

The last activity that must be done to make an internal module work, is including it into the environment. To do this, at the end of constructor code of application (in the file `Application.cc`) the following line should be added:

```
this->modules->push_back((ModuleGeneric *) new Math(variable_space,
     sql));
```

Listing 24: changes in file `Application.cc`.

5.3 External modules

A detailed description of external modules can be found in Subsection 3.4. External modules inherit from `ModuleExternal` class, which is analogous to the class `ModuleInternal`. The only method that needs an implementation is the constructor - all relations and facts occurring in the module must be implemented inside it. Afterwards, file `Application.cc` must be changed - an instance of developed external module must be instantiated.

Constructor

The constructor for external modules has two main goals. The first is to collect all the parameters that the parser encountered in the declaration of the outdoor unit in the 4QL program file. The second one is to generate all relations and facts (external modules do not have any rules nor domains). An exemplary constructor for module `XMLReader` which deals with loading XML format files is provided in Listing 5.3.

```
XMLReader::XMLReader(std::string _name, std::string _filename,
           SQLWrapper* _sql, Checker *_checker, VariableSpace *_vs)
           : ModuleExternal(_vs, _sql) {
    this->name = _name;
    this->checker = _checker;
    this->facts = new std::vector<Fact*>();
    this->relations = new std::vector<Relation*>();
    this->read_file(_filename);
}
```

The method `read_file` deals with loading relations and facts from an XML file. Its implementation is too large to present it here, but the source code can be found in `modules/XMLReader.cc`.

Adding External Modules into the Interpreter Environment

After implementing an external module class, an entry must be added into `Application.cc` file - in method `create_external` an instance of new module must be created. For an exemplary source code for module `xml` see Listing 25.

```
ModuleGeneric *Application::create_external(ExternalDeclaration* e)
     {
    if (e->get_type() == std::string("xml"))
        return this->create_external_xml(e->get_name(), e->
            get_params());
    throw new Exception(std::string("Cannot create module ") +
        e->get_name() + " (type: " + e->get_type() + ")!\n");
}
```

Listing 25: exemplary source code for module `xml`.

The method `create_external_xml` needs to check whether given parameters are correct. It can be done as observed on listing 26.

```
ModuleGeneric *Application::create_external_xml(std::string name,
      std::vector<Value*> *domains) {
   if (domains->size() != 1)
      throw new Exception(std::string("Wrong number of parameters"
         " specified for external module ") + name + "!\n");
   if (domains->at(0)->get_type() != variable_type::STRING)
         throw new Exception(std::string("Wrong type of parameter
             specified "
            "for external module ") + name + "!\n");
   return (ModuleGeneric *) new XMLReader(name,
      domains->at(0)->get_value_string(), this->sql,
      this->checker, this->variable_space);
}
```

Listing 26: checking the correctness of parameters.

5.4 Optimizations

This section describes all the improvements to the project that were found during the design of Inter4QL but have not been implemented in its current version.

Creating the Reasoning Tree

In the current version of the interpreter, each phase of reasoning is finished when none of the rules generates additional facts. In the case when at least one of the rules gives a new fact, a special flag is set. The flag causes all the rules to check for new facts again. It can be noticed immediately that this approach is not very efficient. In the worst case, all rules must be checked even if none of them have the newly generated fact inside their bodies, thus cannot produce any additional facts!

The solution to this problem would be to create an additional data structure in each module containing rules. Let us call it the *reasoning tree*. This structure must share, for each relation name, a list of rules, that have that relation inside their bodies at least once. For example, this structure may be a standard library map from strings (relation names) into the domain of rules, defined as follows:

```
std::map<std::string, std::vector<Rule *>*> links;
```

This structure would fill up during the creation of the module. Reasoning would be a FIFO (first in, first out) queue with all the rules on the beginning. After each rule check, all the rules from the reasoning tree that are not in the queue should be added at the end of the queue. That would prevent checking rules that cannot give any more facts upfront.

Using SQL Queries in Some Functions of Internal Modules

Some of the possibilities offered by the internal modules (e.g., `math` module) can be achieved inside SQL language queries without the need of joining additional database tables. Instead of creating tables in the database, which offers simple operation, these relationships could take part in SQL queries.

One of the relationships that can be easily translated into an SQL query is the relation "greater than" which is provided by the `math` module. Let us recall an example from Section 2.1:

```
tallBoy(A) :- boy(A), hasHeight(A,B), math.ht(B,180)
```

Instead of being translated into the complex query:

```
select t1.param1 from boy t1, hasHeight t2, math_gt t3 where
  t1.param1 = t2.param1 and t2.param2 = t3.param1 and t3.param2 =
      180;
```

it can be translated into a simpler query:

```
select t1.param1 from boy t1, hasHeight t2 where
  t1.param1 = t2.param1 and t2.param2 > 180;
```

That query is simplified by deleting one table join on database relations. This seems not to be a significant difference, but table `math_gt` has n^2 rows (when assuming variable space has n integers) which needs to be calculated before the algorithm starts. That is making a real difference even for not that big n.

A 4QL Language Grammar

This appendix provides a fully described grammar of 4QL language in *Extended Backus-Naur Form*. Nonterminal symbols are marked between brackets, while expressions in monospace font surrounded by quotes are terminal symbols. The symbol ϵ is used for marking empty production.

$\langle program \rangle$ ::= $\langle external \rangle \langle modules \rangle$

```
⟨external⟩                ::= ϵ | ‘external:’⟨external_declarations⟩

⟨external_declarations⟩   ::= ⟨external_declarations⟩⟨external_declaration⟩

⟨external_declaration⟩    ::= ⟨id⟩⟨id⟩‘(’⟨values⟩‘).’

⟨modules⟩                 ::= ϵ | ⟨modules⟩ ⟨module⟩

⟨module⟩                  ::= ‘module’⟨id⟩‘:’⟨domains⟩⟨relations⟩⟨rules⟩⟨facts⟩‘end.’

⟨domains⟩                 ::= ϵ | ‘domains:’ ⟨domain_declarations⟩

⟨domain_declarations⟩     ::= ϵ | ⟨domain_declarations⟩⟨domain_declaration⟩

⟨domain_declaration⟩      ::= ⟨type⟩⟨id⟩‘.’

⟨relations⟩               ::= ϵ | ‘relations:’ ⟨relation_declarations⟩

⟨relation_declarations⟩   ::= ϵ | ⟨relation_declarations⟩⟨relation_declaration⟩

⟨relation_declaration⟩    ::= ⟨id⟩‘(’⟨parameters⟩‘)’

⟨rules⟩                   ::= ϵ | ‘rules:’⟨rule_declarations⟩

⟨rule_declarations⟩       ::= ϵ | ⟨rule_declarations⟩ ⟨rule_declaration⟩

⟨rule_declaration⟩        ::= ⟨rule_head⟩‘:-’⟨relation_list⟩‘.’

⟨rule_head⟩               ::= ⟨id⟩‘(’⟨arguments⟩‘).’
                            |  ‘-’⟨id⟩‘(’⟨arguments⟩‘).’

⟨relation_list⟩           ::= ⟨relation_in_rule⟩
                            |  ⟨relation_list⟩‘,’⟨relation_in_rule⟩
                            |  ⟨relation_list⟩‘|’⟨relation_in_rule⟩

⟨relation_in_rule⟩        ::= ⟨id⟩‘(’⟨arguments⟩‘)’
                            |  ‘-’⟨id⟩‘(’⟨arguments⟩‘)’
                            |  ⟨id⟩‘.’⟨id⟩‘(’⟨arguments⟩‘)’
                            |  ‘-’⟨id⟩‘.’⟨id⟩‘(’⟨arguments⟩‘)’
                            |  ⟨id⟩‘(’⟨arguments⟩‘) in {’⟨logic_values⟩‘}’
                            |  ‘-’⟨id⟩‘(’⟨arguments⟩‘) in {’⟨logic_values⟩‘}’
                            |  ⟨id⟩‘.’⟨id⟩‘(’⟨arguments⟩‘) in {’⟨logic_values⟩‘}’
                            |  ‘-’⟨id⟩‘.’⟨id⟩‘(’⟨arguments⟩‘) in {’⟨logic_values⟩‘}’

⟨logic_values⟩            ::= ⟨logic⟩ | ⟨logic_values⟩‘,’⟨logic⟩

⟨arguments⟩               ::= ⟨argument⟩
                            |  ⟨arguments⟩⟨argument⟩
```

⟨*argument*⟩	::=	⟨*value*⟩⟨*variable*⟩
⟨*facts*⟩	::=	ϵ | ‘`facts:`’⟨*fact_declarations*⟩
⟨*fact_declarations*⟩	::=	ϵ
	|	⟨*fact_declarations*⟩⟨*fact_declaration*⟩
⟨*fact_declaration*⟩	::=	⟨*id*⟩‘`(`’⟨*values*⟩‘`).`’
	|	‘`-`’⟨*id*⟩‘`(`’⟨*values*⟩‘`).`’
⟨*values*⟩	::=	⟨*value*⟩
	|	⟨*values*⟩‘`,`’⟨*value*⟩
⟨*value*⟩	::=	⟨*id*⟩ | ⟨*string*⟩ | ⟨*integer*⟩ | ⟨*real*⟩
	|	⟨*logic*⟩ | ⟨*date*⟩ | ⟨*datetime*⟩
⟨*type*⟩	::=	‘`literal`’|‘`string`’|‘`integer`’|‘`real`’
	|	‘`logic`’|‘`date`’|‘`datetime`’
⟨*parameters*⟩	::=	⟨*id*⟩ | ⟨*parameters*⟩ ‘`,`’ ⟨*id*⟩

Non-terminals that have not been specified above are respectively:

- *id* – alphanumerical string that begins with a lowercase letter
- *string* – string surrounded by quotes
- *integer* – integer number (with a minus sign in front or without)
- *real* – number (dot is used as decimal mark)
- *logic* – a word from the set {`true`, `false`, `incons`, `unknown`}
- *date* – date in format `YYYY-MM-DD`
- *datetime* – date with hours in format `YYYY-MM-DD HH-II`

B Interpreter Command Line

This appendix presents all the command line commands of Inter4QL, described in more detail in Subsection 4.2.

- `import "program.4ql".` - causes the interpreter to load the program contained in file `program.4ql`
- `print module.` - print outs the basic information about module `module`
- `save "database.db".` - saves entire knowledge base in *SQLite3* file format `database.db`

- `savedb "database.db".` - same as above
- `save module "module.xml".` - saves entire module well-supported model in XML file `module.xml`
- `savexml module "module.xml".` - same as above
- `module.relation(param1, param2).` - causes a query for tetravalent value of fact `relation(param1, param2)` inside module `module`
- `modules.` - lists all the modules available in the environment
- `quit.` - exits the interpreter.

C 4QL programs examples

```
module m0:
  relations:
    a(literal).
  rules:
    a(wait) :- a(overloaded) | a(resttime).
    a(resttime) :- a(wait).
    -a(overloaded) :- a(resttime).
    a(goodmood) :- a(rested) | a(success).
    -a(rested) :- -a(resttime).
  facts:
    a(overloaded).
    a(rested).
    a(success).
end.
```

Listing 27: examplary 4QL program.

The well-supported model for program from Listing 27 is:

{`a(success)`, `a(good_mood)`, `a(overloaded)`, `-a(overloaded)`, `a(wait)`, `-a(wait)`, `a(rest_time)`, `-a(rest_time)`, `a(rested)`, `-a(rested)`}

```
module data:
  domains:
    literal name.
    integer height.
  relations:
    canReach(name).
    hasHeight(name, height).
    boy(name).
    tallBoy(name).
  rules:
    canReach(A) :- tallBoy(A).
    tallBoy(A) :- boy(A), hasHeight(A,B), math.gt(B,185).
    -tallBoy(A) :- boy(A), hasHeight(A,B), math.gt(186,B).
  facts:
    boy(tomek).
    -tallBoy(tomek).
    hasHeight(tomek, 190).
end.
```

Listing 28: exemplary 4QL program.

The well-supported model for program from Listing 28 is:

{`boy(tomek), hasHeight(tomek, 190), tallBoy(tomek), -tallBoy(tomek), canReach(tomek), -canReach(tomek)`}

D The Structure of XML Files Interpreted by Inter4QL

Inter4QL gives user the ability to load an XML file with the data (as an external module - a full description can be found in Section 3.2). This appendix describes the correct structure of an XML file that can be loaded by the class XMLReader.

External data source in the form of an XML file must define the relationships that occur in it, and a collection of facts. The main element in the structure of the module must be named and must contain two elements: relations and facts. Below is a diagram of the structure of the file:

```
<module>
    <relations>
        <!-- relation definitions section -->
    </relations>
    <facts>
        <!-- fact definitions section -->
    </facts>
</module>
```

Defining Relations

Definitions of relations are inside the element `relations` and each of them is an independent element of `relation` type. The relationship is defined by setting the name (in an element called `name`) and to determine typical parameters (`params` element inside).

Types of parameters are defined by a list of `param` elements, each of which contains one of the built-in Inter4QL data types (full list can be found in Subsection 2.1). An example of the definition of a relation is shown below:

```
<relation>
    <name>boy</name>
    <params><param>literal</param></params>
</relation>
```

Defining Facts

Definitions of facts are inside the element `facts` and each of them is an independent element of `fact` type. Name of the relation must be provided (in element `name`) which fact points to, and a list of parameters (in element `params`, defined the same way as in the relation definition). If a fact is negative there must be a `negated` empty XML entity inside. Examples of fact definitions can be found in Listing 29.

```
<fact>
    <name>boy</name>
    <params><param>tomek</param></params>
</fact>
<fact>
    <negated/>
    <name>boy</name>
    <params><param>marcelina</param></params>
</fact>
```

Listing 29: exemplary definitions of facts.

Example of a Valid XML File

```
<module>
  <relations>
    <relation>
      <name>hasHeight</name>
      <params><param>literal</param><param>integer</param></params>
    </relation>
    <relation>
      <name>boy</name>
      <params><param>literal</param></params>
    </relation>
  </relations>
  <facts>
    <fact>
      <name>boy</name>
      <params><param>tomek</param></params>
    </fact>
    <fact>
      <name>hasHeight</name>
      <params><param>tomek</param><param>190</param></params>
    </fact>
  </facts>
</module>
```

References

[1] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison-Wesley Pub. Co., 1996.

[2] J. Maluszyński, A. Szałas, *Living with Inconsistency and Taming Non-monotonicity*, in: Datalog Reloaded, G. Gottlob, G. Grasso, O. de Moor, and A. Sellers, eds., LNCS 6702, 384–398, 2011.

[3] J. Maluszyński, A. Szałas, *Logical Foundations and Complexity of 4QL, a Query Language with Unrestricted Negation*, Journal of Applied Non-Classical Logics 21(2), 211–232, 2011.

[4] P. Spanily, *Interpreter for four-valued rule-based query language 4QL*, M.Sc. Thesis, Institute of Informatics, University of Warsaw, 2011. In Polish.

[5] *Bison - GNU parser generator*, http://www.gnu.org/software/bison/

[6] *Flex: The Fast Lexical Analyzer*, http://flex.sourceforge.net/

[7] *TinyXML C++ library for parsing XML*, http://sourceforge.net/projects/tinyxml/

[8] *Doxygen*, http://www.stack.nl/∼dimitri/doxygen/

[9] *SQLite3 Home*, http://www.sqlite.org/

[10] *MinGW – Minimalist GNU for Windows*, http://www.mingw.org/

[11] *About 4QL at 4QL*, http://4ql.org/